A model-based design-for-verification approach to checking for deadlock in multi-threaded applications

Beata Sarna-Starosta, R. E. K. Stirewalt, and Laura K. Dillon
Software Engineering and Network Systems Laboratory
Dept. of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824

Abstract

This paper explores an approach to design for verification in systems that are built atop a middleware framework designed to separate synchronization concerns from the "core-functional logic" of a program. The framework is based on a language-independent compositional model of synchronization contracts, called Szumo, which integrates well with popular OO design artifacts and provides strong guarantees of non-interference. An approach for extracting models from Szumo design artifacts and analyzing the generated models to detect deadlocks is described. A key decision was to use Constraint Handling Rules to express the semantics of synchronization contracts, which allowed a transparent model of the implementation logic.

1. Introduction

A key problem in the verification of concurrent software is the need to extract from design artifacts a *finite state model* that is amenable to analysis with respect to a property of interest. Approaches under the category *design for verification* (D4V) concede some degree of freedom during design (e.g., prescribing design rules [15], or requiring designers to instantiate a predefined set of patterns [12, 5]) to automate the derivation of such models. This paper explores an approach to D4V in systems that are built atop a middleware framework designed to separate synchronization concerns from the "core-functional logic" of a program.

The SynchroniZation Units MOdel (Szumo) framework associates each thread with a *synchronization contract* that governs how it must synchronize with other threads. At run time, schedules are derived by *negotiating* contracts among threads, so that a thread is scheduled only if its contract has been successfully negotiated. Applications using Szumo execute atop a middleware layer that implements the details of negotiation and scheduling. The contracts themselves are formed by conjoining module-level *synchronization constraints*, which a programmer declares in the module's interface. A novel characteristic of Szumo is that programmers declare these constraints in lieu of writing explicit synchronization code in the module's implementation, i.e., deferring the mechanics of synchronization to the middleware.

The middleware negotiates contracts in a manner that guarantees freedom from a large class of data races [13] and deadlocks. Consequently, these properties need not be verified. However, a model of the application is required to demonstrate freedom from all forms of deadlock and to verify more general safety properties. This paper describes an approach that leverages the separation of concerns provided by the Szumo framework to simplify extraction of models from a design that will be implemented using Szumo. The generated models are analyzed to detect deadlock.

The remainder of this paper describes our D4V approach. We build finite-state models from UML diagrams extended to represent synchronization constraints (Section 2). A critical component is the verifier customized to solve systems of synchronization constraints (Section 3). We found it natural to specify the synchronization semantics in the language of Constraint Handling Rules (CHR). We conclude with a summary of our approach and its comparison with related work (Section 4).

2. Background

We designed Szumo to support development of multi-threaded object-oriented (OO) systems, for which a key problem is to synchronize threads that operate over shared data. Thread synchronization logic is inherently complex, as it involves reasoning over state spaces that grow non-linearly with the size of the program. Without proper synchronization, concurrent access to shared objects can lead to race conditions, and incorrect synchronization logic can lead to starvation or deadlock. Szumo is a

language-independent model of synchronization contracts and middleware-based contract negotiation [3].[1] To date, we have integrated it into an extension of the Eiffel language and, more recently, as an object-oriented framework in C++. Szumo is the basis for the D4V approach described in the sequel, which implements the verification engine using CHR. This section supplies background on relevant D4V artifacts, Szumo and CHR.

2.1. D4V Artifacts

A fundamental D4V decision involves the granularity of sharing among threads: To improve the efficiency of analysis, a designer may opt for coarse-grain sharing; however, doing so limits concurrency. In Szumo, designers choose the granularity of sharing by deploying program objects into *synchronization units*, which are "object-like" containers of one or more program objects. When a program object is created, the middleware deploys it to exactly one synchronization unit, where it remains throughout its lifetime. Moreover, a thread that holds exclusive access to a synchronization unit holds exclusive access to all program objects contained within that unit. A synchronization unit is object-like in that it may encapsulate state, provide operations, and reference other synchronization units whose operations it uses to implement its own, i.e., in a client–supplier fashion. Because sharing occurs only at the level of units, concurrency analyses need not consider how units are composed of program objects. Thus, in the sequel, the "objects" we refer to will always be synchronization units and not program objects.

Synchronization units are identified with instances of types, called *unit classes*. In addition to standard operations, a unit class may declare (zero or more) *unit variables*, *condition variables*, and synchronization constraints. Unit variables are references in a client class C to units that serve as suppliers to units of type C. Condition variables are boolean-valued variables in a unit class. The *synchronization-relevant state* of a unit is represented by an assignment of values to the unit's condition variables and unit variables. Finally, the synchronization constraints declare the suppliers to which a client unit requires exclusive access as functions of the client's synchronization-relevant state. A client unit c is said to *entail* a supplier unit s when c requires exclusive access to s.

To illustrate these ideas and Szumo design artifacts, we show a UML design for a solution to the classic dining philosophers problem (Fig. 1). A *unit-class diagram* (top left) documents unit classes and relationships between them. The stereotypes ⟨⟨`synchronization`⟩⟩ and ⟨⟨`process_root`⟩⟩ designate unit classes whose instances are, respectively, units that are sharable among multiple threads, and units that serve as non-shared "roots" of a given thread. We show condition variables as class attributes, unit variables as directed associations, and synchronization constraints as limited propositional formulae over condition variables and unit variables, formed using an *entailment operator* "`==>`". The set of synchronization constraints associated with a client class is shown adjacent to that class. Thus, in this design, philosopher units execute in different threads and may invoke operations on shared fork units bound to their `left` and `right` unit variables. Associated with each philosopher, condition variable `eating` signifies when the philosopher needs exclusive access to its forks. The philosopher's synchronization constraint asserts that, if `eating` is true, the philosopher entails its left and right forks.

A *sync-state diagram* (Fig. 1, bottom left) shows how operations affect the values of condition variables. The sync-states are annotated with the condition variables they map to true. The transitions carry annotations to designate when they are taken. An arrow with no source sync-state marks the initial sync-state. Thus, `eating` is false in `s0`, the initial sync-state of a philosopher unit, and true in `s1`. When in `s0`, a philosopher transitions into `s1` (immediately) before invoking its `eat` operation, and transitions back to `s0` (immediately) upon returning. Together, the unit-class and sync-state diagrams document that a philosopher entails its forks while executing an `eat` operation.

Unit-class and sync-state diagrams are reusable over many different programs. In contrast, a *unit-instance diagram* (Fig. 1, right) captures a particular configuration of synchronization units representing the initial state of a design to be checked for deadlock.

2.2. Overview of Negotiation

In a Szumo design, the set of units that a thread needs to access can be inferred at run time. For instance, from the design artifacts in Fig. 1, we infer that a thread needs only its root unit, except when executing the root's `eat` operation, when it needs the `Fork` units referenced by the root's `left` and `right` unit variables. More generally, a thread needs its root unit, as well as any unit entailed by a unit that it needs. The conjunction of the synchronization constraints associated with the units that a thread needs defines the thread's synchronization contract. The contract changes dynamically as the units that the thread executes modify condition variables and unit variables.

At run time, the Szumo middleware associates with each thread a set of units, called the thread's *realm*. When a thread is executing, it is allowed to access all units in its realm, and prevented from accessing any units not in its realm. Szumo provides a strong guarantee of non-interference by preventing realms from overlapping—i.e.,

[1] called the "universe model" in [3]

Figure 1. Unit-Class (top left), Sync-state (bottom left), and Unit-Instance (right) Diagrams

by preventing any unit from simultaneously being in the realm of more than one thread. Changes in a unit's synchronization-relevant state may result in a thread's realm containing units that it no longer needs. To maximize concurrency, such units should be *released*, or migrated out of the thread's realm. Changes in a unit's synchronization-relevant state may also result in a thread needing some units that are not in its realm. Such units must be *acquired* and migrated into the realm before the thread can be scheduled. Because some needed units may be contained in the realms of other threads, it may not be possible to immediately acquire all needed units. When a thread's realm contains exactly the set of needed units, we say that the realm is *complete*; otherwise we say it is *damaged*.

From a user's perspective, a thread executes within a complete realm until it performs an operation that changes the synchronization-relevant state of a unit, thereby damaging the realm. For instance, consider a thread whose initial realm contains just a philosopher unit p in sync-state `s0`. Then p's entailment is the empty set and so, according to the thread's contract, this initial realm is complete. However, by invoking the `eat` operation, the thread changes p's sync-state to `s1`, thereby changing its entailment and damaging the realm. The semantics of Szumo dictate that a thread with a damaged realm must block until the realm can be *repaired*. Repairing a damaged realm involves releasing and/or acquiring units to make the realm complete. In our example, the units referenced by p's `left` and `right` unit variables would need to be acquired and migrated into the realm containing p. To safely migrate units, a thread must negotiate with other threads.

In a Szumo program, the middleware enforces and negotiates contracts on behalf of threads. It enforces a thread's contract by confining the thread to execute within its realm.[2] The negotiation of a thread's contract proceeds according to a decentralized two-phase protocol. In the initial *contraction phase*, the middleware releases from the realm all units that are no longer needed. Then, in the *expansion phase*, the middleware attempts to incrementally acquire units not in the realm but in the entailment of some unit in the realm and migrate them into the realm. The expansion phase is complicated by the need to prevent the introduction of concurrency errors in incrementally acquiring units.

[2]it traps accesses by a thread to a unit, raising a run-time exception if the unit is not contained in the realm

2.3. Szumo Negotiation Predicates

For concurrency analysis, we do not model the full mechanics of negotiation, but only the effects of negotiation on the realms and execution status of threads. We define such effects using four predicates. The first predicate represents the entailment information derived from a design: *entails*(u, v) asserts that u entails v, for units u, v. This predicate is used to define completeness: A set of units R is complete for a thread t if it is the least set satisfying

- $r \in R$, where r denotes the root unit of t and
- $u \in R$ and *entails*(u, v) implies $v \in R$, for units u, v

The remaining predicates involve a thread t and a unit u:

- *holds*(t, u) asserts that u is in the realm of t
- *needs*(t, u) asserts that, if a set R is complete for t, then $u \in R$
- *waits*(t, u) asserts that t needs u but that some other thread t' holds u, where $t \neq t'$

When the realm of a thread t is first damaged, the realm may contain units u such that *holds*(t, u), but not *needs*(t, u). Operationally this means that the realm must be contracted. During expansion, any units u satisfying *needs*(t, u), but not *holds*(t, u), must be migrated into the realm. The realm is complete when *holds*(t, u) and *needs*(t, u) are equivalent, for all u. A deadlocking state is one in which there exists a cycle of dependencies between the *waits* and *holds* relations of two or more threads.

2.4. Constraint Handling Rules

Constraint Handling Rules (CHR [8]) is a declarative language extension designed to write application-tailored constraint solvers. It specifies multi-headed guarded rules that re-write constraints until they reach a solved form. The rules define valid transformations of constraints collected in a dynamically changing *constraint store*. When a constraint is *posed*, it is added to the constraint store, and then the rules are applied exhaustively until either no more transformations can be performed, in which case the evaluation succeeds, or the constraint store becomes inconsistent, in which case the evaluation fails.

A CHR rule has the form

```
Label @ Head T Guard | Body.
```

where `Label` names the rule; `Head` identifies stored constraints subject to transformation; `T` signifies the kind of transformation to be performed, either *simplification*, written "`<=>`", or *propagation*, written "`==>`"; `Guard` is a condition that must hold for the rule to apply; and `Body` specifies constraints to be posed.

A rule is applied if the posed constraint matches a constraint in the rule's head, the store contains constraints that match the other head constraints, and the rule's guard is satisfied. A simplification rule requires that all head constraints be removed from the constraint store and the constraints in the body of the rule be posed. A propagation rule requires that the body constraints be posed but the head constraints remain in the store. Thus, simplification replaces some stored constraints with others, while propagation adds to the store new constraints which may cause further simplifications.

Table 1 shows some of the rules that we use in defining the Szumo negotiation protocol (Section 3). `Lift` states that for every pair of stored constraints of the form `entails(U,U1)` and `holds(T,U)`, a constraint `needs(T,U1)` should be posed. `NonInt` applies whenever the store contains `holds(T1,U)` and `holds(T2,U)` such that $T_1 \neq T_2$, in which case the evaluation fails. `Acquire` and `Block` replace stored constraints matching the left-hand side with the constraints indicated by the corresponding right-hand side.

3. Deriving models from designs

We model the behavior of a Szumo system as an *extended finite automaton* (EFA) whose states represent configurations of threads, each executing within a unit in its realm. Transitions of this EFA reflect synchronization-relevant changes in the state of one or more threads and thus must respect the negotiation semantics of unit migration as dictated by synchronization constraints. This semantics is obtained by extending the EFA-based verification framework of [14] to support inter-thread negotiation.

```
Lift @ entails(U,U1),holds(T,U)
       ==> needs(T,U1).
Acquire @ needs(T,U) <=> holds(T,U).
NonInt @ holds(T1,U),holds(T2,U)
          <=> T1≠T2|fail.
Block @ needs(T,U) <=> waits(T,U).
```

Table 1. CHR for thread negotiation

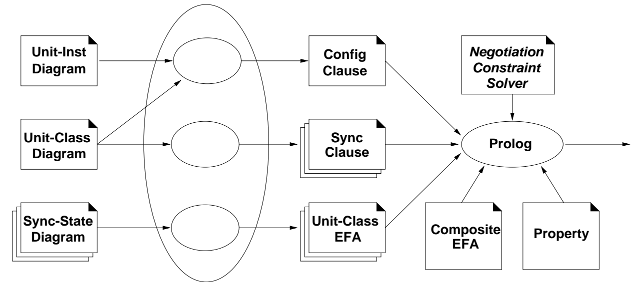

Figure 2. Tools and artifacts in our process

Fig. 2 outlines the high-level architecture of our D4V framework. Szumo design artifacts (far left) are translated into Prolog modules, which instantiate a generic deadlock-detection framework with the details of the system to be analyzed. Below we describe each of the generated Prolog modules. As of the time of writing, the translation (unlabelled ovals) from Szumo design artifacts to Prolog modules is performed by hand.

3.1. Problem-specific framework modules

For each unit class, a *Unit-Class EFA* module defines two relations to use in modeling the units' behaviors: `trans` and `inv`. The `trans` relation represents the transition relation defined in the unit class' sync-state diagram. For example, the following rules are generated from the sync-state diagram for a philosopher (Fig. 1)

```
trans(philosopher, P, s0, before_eat, s1).
trans(philosopher, P, s1, after_eat, s0).
```

where `philosopher` names the associated unit class, `P` identifies a specific unit-class instance, and `s0`, `s1` and `before_eat`, `after_eat` correspond to sync-state labels and transition labels, respectively.

The `inv` relation associates each sync-state with an *invariant* specifying the values of the unit's condition variables in the sync-state. For example,

```
inv(philosopher, s0, [], [eating]).
inv(philosopher, s1, [eating], []).
```

signifies that, when a philosopher unit is in `s0`, `eating` is false, and when the unit is in `s1`, `eating` is true.

Each synchronization constraint is translated into a *Sync(hronization) Clause* which is used in computing a unit's entailments. For example, "`eating ==> left` $\wedge$ `right`" yields

```
synch(P,eating,[L,R]) :-
  left(P,L), right(P,R).
```

where `P` identifies a philosopher unit, `L` and `R` reference the unit's left and right fork, and `left` and `right` are CHR constraints, posed here to retrieve the units bound to the corresponding unit variables.

The unit-instance diagram manifests as a *Config(uration) Clause* which records the inter-unit associations through unit variables. For example, the unit-instance diagram in Fig. 1 translates into

```
left(P0,F1).  left(P1,F2).  left(P2,F0).
right(P0,F0). right(P1,F1). right(P2,F2).
```

3.2. Generic framework modules

Unit-level EFAs are aggregated into thread-level EFAs, which are then composed to form a global EFA. The composition is performed by the *Composite EFA* module, which computes successor states in the global EFA based on the successor states in the thread EFAs and the protocol of inter-thread negotiation defined by the *Negotiation Constraint Solver* (Section 3.3).

A transition in the global EFA is constructed from a single *execution step* in the unit-class EFA of a distinguished *active* unit. An execution step is a unit-level transition into a new sync-state followed by an update of the active unit's condition variables in accordance with the unit's `trans` and `inv` relations. These updates trigger the addition and/or removal of `entails` constraints to reflect changes in the unit's entailment.

The *Property* module checks each newly computed system state for deadlock. It searches the constraint store for a set of constraints defining a cycle of *holds* and *waits* dependencies.

3.3. Negotiation Constraint Solver

A change to the entailment of a unit may damage the realm of the thread that owns it. Repairing a damaged realm requires inter-thread negotiation. The *Negotiation Constraint Solver* is a Prolog module reflecting the semantics of negotiation. It is used at each step of system execution to update constraints representing the predicates in Section 2.3.

We generate the negotiation constraint solver from a CHR program. Table 1 shows some of the rules in this program. Intuitively, the `Lift` rule poses `needs(T,`U_i`)` for each stored `entails(U,`U_i`)`. In posing `needs(T,U)`, the `Acquire` rule attempts to replace `needs(T,U)` with `holds(T,U)` signifying that `U` is in `T`'s realm. However, if the constraint store already contains a constraint `holds(`T'`,U)` for some thread $T' \neq T$, the `NonInt` rule causes this attempt to fail. In this case, the `Block` rule replaces `needs(T,U)` with `waits(T,U)`, signifying that `T` blocks waiting for `U`.

4. Discussion and Related Work

This paper builds upon the separation of concerns and run-time guarantees provided by Szumo and the declarative nature of CHR to enable an approach for detecting deadlock in designs of multi-threaded systems. Our D4V approach extends the EFA-based verification system of [14] to simplify extraction of finite state models from Szumo designs. In Szumo designs, synchronization concerns are separated from the "core-functional logic" of programs and expressed as synchronization contracts, which are to be automatically negotiated at run time by middleware. The verification system is extended with a constraint solver that is generated from a CHR program encoding the semantics of synchronization contracts. The simplicity of this encoding contributes to the transparency of our approach and facilitates traceability between designs and their models. Additionally, the verification system constructs models on-the-fly [10], elaborating execution paths incrementally and only to the point where either a deadlock is detected or extending a path would violate a synchronization contract. Although not discussed in this paper, our D4V approach also supports checking of more general properties than just deadlock by substituting appropriate property modules in Fig. 2.

Concurrency analysis tools such as Bandera [6] and Java PathFinder (JPF) [9] extract finite state models directly from code, instead of designs, to ensure fidelity between the code and the model. JPF translates programs written in a Java subset to PROMELA in order that the SPIN model checker [10] can detect deadlocks and violations of boolean assertions. Bandera incorporates a variety of program analysis, abstraction, and transformation techniques for extracting conservative finite-state models from Java source code, and a variety of backends targeting input languages of different model checkers. The translations implemented by JPF and Bandera are complex, obstructing transparency and complicating traceability. The models automatically generated by these tools also tend to be fairly low-level. To counter this latter problem, Bandera relies on a software analyst to indicate additional abstractions it should perform. In contrast, our D4V approach generates models from de-

sign artifacts that express synchronization concerns declaratively and at a higher level than code. Such artifacts should be more amenable to transparent generation of models at a level of abstraction suitable for concurrency analysis.

Several other approaches separate synchronization concerns for purposes of verification. One notable approach, SyncGen, generates synchronization code that is woven into a subject program from declarative specifications of *region invariants* [7]. A second, synthesizes *concurrency controllers* to coordinate threads in accessing protected resources from global policy specifications, expressed using high-level guarded commands based on a set of pre-defined patterns, and controller interface specifications, expressed as finite state machines [4]. In both approaches, the separation of concurrency concerns affords more modular reasoning. However, the specifications of concurrency concerns in these approaches are global in that they describe possible accesses made by all threads to supplier objects. As such, they are more expressive than, but they lack the compositionality of synchronization contracts. Furthermore, neither of the approaches provides protection from data races if an error is made in specifying concurrency concerns and neither directly supports analysis for deadlocks.

Magee and Kramer propose a model-based approach to D4V that separates synchronization and functional concerns [11]. The designer first models a system as the parallel composition of finite state processes (FSP) and analyzes the model for safety and progress properties using their LTSA tool. Once verified, the processes are implemented as monitors in Java. The translation from an FSP model to Java is idiomatic, not automatic. Additionally, the FSP model is intentionally operational.

An approach that supports D4V extends Java extends Java with annotations to associate locks with program fields and methods in order to support the detection of race conditions [1]. The approach uses static analysis to track the set of locks held at each program point and supports both client- and supplier-side synchronization. Annotations are supplied by a designer or inferred automatically, which, for the price of false positives, makes the technique applicable even for very large programs. Our approach, by contrast, uses analysis to detect deadlock leaving protection against data races to the middleware.

Finally, we note that others have used CHR to reason about concurrent interactions. Alberti and colleagues verify that multi-agent configurations behave according to prescribed protocols [2]. However, they do not use constraint programming to directly model the semantics of agent interaction, as we use it to model the semantics of thread negotiation.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[2] M. Alberti et al. Specification and verification of agent interaction protocols in a logic-based system. In *Proc. ACM Symp. Applied Computing*, pages 72–78, 2004.

[3] R. Behrends and R. E. K. Stirewalt. The Universe Model: An approach for improving the modularity and reliability of concurrent programs. In *Proc. ACM SIGSOFT Symp. Foundations Softw. Eng.*, 2000.

[4] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proc. Conf. Automated Softw. Eng.*, pages 248–257, 2004.

[5] T. Bultan and A. Betin-Can. Scalable software model checking using design for verification. In *Proc. IFIP Working Conf. Verified Softw.: Theories, Tools, Experiments*, 2005.

[6] J. C. Corbett et al. Bandera: extracting finite-state models from java source code. In *Proc. Intl. Conf. Softw. Eng.*, pages 439–448, 2000.

[7] X. Deng et al. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proc. Intl. Conf. Softw. Eng.*, 2002.

[8] T. Frühwirth. Theory and practice of constraint handling rules. *Jrnl. Logic Prog.*, 37(1-3):95–138, October 1998.

[9] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Intl. Jrnl. Softw. Tools for Technology Transfer*, 2(4), Apr. 2000.

[10] G. Holzman. *The SPIN Model Checker*. Addison-Wesley Inc., 2004.

[11] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 2000.

[12] P. Mehlitz and J. Penix. Design for verification using design patterns to build reliable systems. In *Proc. ICSE Workshop Component-Based Softw. Eng.*, 2003.

[13] R. H. B. Netzer and B. P. Miller. What are race conditions?: Issues and formalizations. *ACM Letters Prog. Lang. and Systems*, 1(1):74–88, Mar. 1992.

[14] B. Sarna-Starosta and C. Ramakrishnan. Constraint-based model checking of data-independent systems. In *Proc. Intl. Conf. Formal Eng. Methods (ICFEM)*, pages 579–598, 2003.

[15] N. Sharygina, J. C. Browne, and R. P. Kurshan. A formal object-oriented analysis for software reliability: Design for verification. In *Proc. FASE*, pages 318–332, 2001.